AN-CA-001
Floating Point Multiplication on ARM

J. Haas, J. Frederickson, J. Liu, G. Umlauf
DEC 2013

Overview

This application note will analyze a floating point multiplication operation on an ARM processor. The assembly code generated using a C++ compiler will be examined and explained to familiarize the reader with ARM and floating point calculations. This report draws heavily on the work of Anthony Merlino, currently teaching the Introduction to Computer Architecture laboratory section at Rowan University.

Background

ARM is an instruction set used by billions of electronic devices today. It is a Reduced Instruction Set Computing (RISC) architecture developed and maintained by ARM Holdings. The most recent version is ARMv8-A, which can be used for 32-bit or 64-bit applications. Among other requirements, the specification includes 31 general purpose registers.

The idea of floating point arithmetic was created to address the limitations of integers in computers. Namely, floating point numbers have a decimal component, and can represent values many orders of magnitude larger and smaller than can be achieved with integers. IEEE 754, the Standard for Floating-Point Arithmetic, defines the methodology used by virtually every computer on the market today to represent and manipulate floating point numbers [1].

Representing a number using IEEE 754 breaks it into three parts: the sign bit, the exponent, and the significand. This paper will be discussing single precision numbers represented using 32 bits. The leftmost bit is the Sign Bit, where 0 indicates positive and 1 indicates negative. The next 8 bits make up the exponent, and the final 23 bits the significand. The exponent is biased by 127; in other words, when the exponent is stored it is first added to 127. Thus, instead of values from -127 to 128, the exponent can be from 0 to 255. Finally, there is an assumed 1 bit to the left of the significand, resulting in 24 effective bits of storage.

Problem

How does floating point multiplication work in the ARM instruction set?

Approach

This report used the BeagleBone Black (Figure 1) to compile and execute code. The Black has a 1GHz AM335x ARM Cortex-A8 TI Sitara Microprocessor. In order to view the disassembly of a C++ program, we used Mr. Anthony Merlino's tutorial: “Emulated Floating Point Addition on the Beaglebone” [2]. His setup consists of TI's Code Composer Studio and Starterware for AM335x that uses the BlackHawk XDS100VS Emulator to implement JTAG debugging on the BeagleBone Black. This allowed us to directly view and step through the assembly code for a C++ program.

Figure 1. The BeagleBone Black development board.

Results

The C++ source file was written to multiply two numbers, namely 1.4 and 0.042, and store the result in a third variable. The full source code can be found in Appendix A. Code Composer was then used to view the disassembly presented by the JTAG emulator. The full disassembly can be viewed in Appendix B. The body of the report, however, will only look at the code pertaining to the example multiplication.

Discussion

What follows is a step-by-step analysis of the multiplication operation. First a few lines of assembly code are presented, then an explanation is given following.

The main program:

```
LDR R12, $C$FL1
STR R12, [R13]
```

First, the values are stored in memory. CFL1 is a substitution used by the assembler, but in essence these two commands are storing the value 1.4 to the location in memory pointed to by R13.

```
LDR R12, $C$FL2
STR R12, [R13, #4]
```

The same is done with 0.042, except it is stored 4 bytes (32 bits) after the location pointed to by R13.

```
LDR R0, [R13]
LDR R1, [R13, #4]
BL __aeabi_fmul
```

Now z = x * y; executes. Our first number is stored in R0, and our second number is stored in R1. Next we branch to the location in memory pointed to by __aeabi_fmul, and store a link to the current address in R14 so we can return later.

```
STR R0, [R13, #8]
```

The answer calculated in __aeabi_fmul (see below) is stored in memory 8 bytes (64 bits) after our first number.

Branch __aeabi_fmul:

```
STMFD R13!, {R2, R3, R4, R5, R6,
R14}
```

This command stores the contents of the registers we will be modifying to memory at R13. That way when we finish the __aeabi_fmul branch and return to the main program, the Register File will be virtually unchanged.

```
EORS R6, R0, R1
```

Here we are taking the exclusive or (XOR) of our two numbers and storing the result is R6. The only thing we actually care about is the leftmost bit, which will be the XOR of our numbers' sign bits. The “S” on the end of “EOR” indicates that the Negative (N), Zero (Z), Carry (C), and Overflow (V) flags will be set based on the result.

```
MOVMI R6, #-2147483648
```

This command only executes if the N flag is 1. The N flag takes on the value of the leftmost bit of R6. So if the N flag is 1, that means our numbers had different signs. Therefore, we set the sign bit of R6 to be negative. In two's complement representation, -2147483648 is written as the following: 10000000000000000000000000000000. This does not execute for our example.

```
MOVPL R6, #0
```

If the N flag is 0, then our numbers had the same sign, and the resulting sign must be positive. Therefore, we set the sign bit of R6 to 0.

```
MOV R2, R0, LSL #8
```

This sets R2 to be the LSB of the exponent, then the significand, then 8 zeros. One bit to the left of the significand is necessary for a following step.

```
MOV R3, R0, LSL #1
MOVS R3, R3, LSR #24
```

Here we remove the sign bit from the first number by shifting left once, then remove the significand by shifting right 24 times. R3 is now the exponent of the first number. Also, NZV are updated.

```
ORRNE R2, R2, #-2147483648
```

This adds the assumed bit back into the significand. OR with 10000000000000000000000000000000 guarantees that the first bit will be 1, while all other bits will remain unchanged. This command only executes if the exponent is not 0. If the exponent or significand are 0, we don't want to change anything about them, so zero will be very easy to identify down the line.

```
MOVEQ R0, #0
LDMEQFD R13!, {R2, R3, R4, R5, R6,
PC}
```

If the exponent is zero, by IEEE 754 the entire number must be zero. Note that the actual exponent is 0 – 127 = -127. Our answer is in R0, and we reload the registers we saved earlier and exit the branch back to the main program. This is not executed for our example.

```
CMP R3, #255
BEQ ovfl
```

Here we compare R3 and 11111111 (the max possible exponent) to check for infinity, which IEEE 754 defines as an exponent of 255. Note that the actual exponent is 255 – 127 = 128. If it is infinity, we branch to ovfl. This is not executed in our example.

```
MOV R4, R1, LSL #8
MOV R5, R1, LSL #1
MOVS R5, R5, LSR #24
ORRNE R4, R4, #-2147483648
MOVEQ R0, #0
LDMEQFD R13!, {R2, R3, R4, R5, R6,
PC}
CMP R5, #255
BEQ ovfl
```

These commands correspond exactly to the ones executed for our first number, but this time apply to our second number.

The state of the registers at this point is:

R0: our first number
R1: our second number
R2: first mantissa followed by zeros
R3: zeros followed by first exponent
R4: second mantissa followed by zeros
R5: zeros followed by second exponent
R6: our result's sign bit followed by zeros

```
ADD R3, R3, R5
```

Add the exponents and store the result in R3.

```
UMULL R5, R4, R2, R4
```

Multiply the mantissas and store the 32 LSB in R5, and the 32 MSB in R4. The ones we care about are the MSB in R4.

```
CMP R4, #0
MOVPL R4, R4, LSL #1
SUBPL R3, R3, #1
```

Compare the 32 MSB of the mantissa with 0. Note that compare always updates NZCV. If the 32 MSB are greater than 0, remove the assumed bit from the mantissa and subtract 1 from the exponent.

```
ADDS R4, R4, #128
```

Add 128 = 10000000 to the significand and update NZC. This adds one to the 7th bit, which is the first bit after the leftmost 24 bits (i.e. the first bit our final significand cannot hold). If it was 0, our result is unchanged. If it was 1, the addition effectively rounds up our answer.

```
ADDCS R3, R3, #1
MOVCS R4, R4, LSR #1
```

If a carry resulted from our rounding operation, we have to add 1 to the exponent and shift the mantissa to the right 1 space so as to not lost the overflow. Note that the assumed bit was already removed before this step, so if a carry results 10 must be to the left of our significand. Therefore, shifting right 1 sets the MSB to 0 and we retain the assumed 1. This is not executed in our example.

```
SUBS R3, R3, #126
```

Keep in mind that we never converted the exponents back from excess 127. Therefore, they are both 127 larger than they need to be. So we subtract 126 from the exponent and update NZCV. Combined with “SUBPL R3,R3,#1” above, this subtracts 127 from the exponent. This will be the final biased exponent, since it is the same as (e1+127) + (e2+127) – 127 = (e1+e2) + 127.

```
MOVLE R0, #0
LDMLEFD R13!, {R2, R3, R4, R5, R6,
PC}
```

If the exponent was less than or equal to 126 (i.e. R3 is now <=0 after subtracting 126), the result is 0 because an exponent of 0 or lower is equal to 0 by IEEE 754. Then load registers and exit to main. This is not executed in our example.

```
CMP R3, #255
BCS ovfl
```

Compare the exponent and 11111111 (max possible exponent). If the exponent is greater than 255, it is infinity by IEEE 754 so we branch to ovfl. This is not executed in our example.

```
MOV R0, R4, LSR #8
```

Put the significand in R0 and make room for the exponent by shifting right 8 spaces.

```
BIC R0, R0, #8388608
```

This is a Bit Clear operation. It performs AND on R0 and the 1's complement of 8388608, which is 00000000011111111111111111111111. It sets all the bits to 0 except the significand, which is unchanged. This is done because we originally had our first number stored in R0.

```
ORR R0, R0, R3, LSL #23
```

This places the exponent next to the significand in R0 in the proper place.

```
ORR R0, R0, R6
```

This sets the sign bit in R0.

```
LDMFD R13!, {R2, R3, R4, R5, R6,
PC}
```

Finally we reload the registers and return to the main program with our result in R0!

References

[1] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008* , vol., no., pp.1,70, Aug. 29 2008. DOI: 10.1109/IEEESTD.2008.4610935

[2] *ARM Architecture Reference Manual*, I ed., ARM Limited, Cambridge, England, 2005.

[3] A. Merlino. (2013). *Emulated Floating Point Addition on the Beaglebone* [Online]. Available: http://anthonymerlino.us/uncategorized/emulated-floating-point-addition-on-the-beaglebone

Appendix A – Source Code

```
int main(void) {
float x = 1.4;
float y = .042;
float z;


z = x * y;


return 0;
}
```

Appendix B – Disassembly

```
          __aeabi_fmul:
800008bc:   E92D407C STMFD          R13!, {R2, R3, R4, R5, R6, R14}
800008c0:   E0306001 EORS           R6, R0, R1
800008c4:   43A06102 MOVMI          R6, #-2147483648
800008c8:   53A06000 MOVPL          R6, #0
800008cc:   E1A02400 MOV            R2, R0, LSL #8
800008d0:   E1A03080 MOV            R3, R0, LSL #1
800008d4:   E1B03C23 MOVS           R3, R3, LSR #24
800008d8:   13822102 ORRNE          R2, R2, #-2147483648
800008dc:   03A00000 MOVEQ          R0, #0
800008e0:   08BD807C LDMEQFD        R13!, {R2, R3, R4, R5, R6, PC}
800008e4:   E35300FF CMP            R3, #255
800008e8:   0A000019 BEQ            ovfl
800008ec:   E1A04401 MOV            R4, R1, LSL #8
800008f0:   E1A05081 MOV            R5, R1, LSL #1
800008f4:   E1B05C25 MOVS           R5, R5, LSR #24
800008f8:   13844102 ORRNE          R4, R4, #-2147483648
800008fc:   03A00000 MOVEQ          R0, #0
80000900:   08BD807C LDMEQFD        R13!, {R2, R3, R4, R5, R6, PC}
80000904:   E35500FF CMP            R5, #255
80000908:   0A000011 BEQ            ovfl
8000090c:   E0833005 ADD            R3, R3, R5
80000910:   E0845492 UMULL          R5, R4, R2, R4
80000914:   E3540000 CMP            R4, #0
80000918:   51A04084 MOVPL          R4, R4, LSL #1
8000091c:   52433001 SUBPL          R3, R3, #1
80000920:   E2944080 ADDS           R4, R4, #128
80000924:   22833001 ADDCS          R3, R3, #1
80000928:   21A040A4 MOVCS          R4, R4, LSR #1
8000092c:   E253307E SUBS           R3, R3, #126
80000930:   D3A00000 MOVLE          R0, #0
80000934:   D8BD807C LDMLEFD        R13!, {R2, R3, R4, R5, R6, PC}
80000938:   E35300FF CMP            R3, #255
8000093c:   2A000004 BCS            ovfl
80000940:   E1A00424 MOV            R0, R4, LSR #8
80000944:   E3C00502 BIC            R0, R0, #8388608
80000948:   E1800B83 ORR            R0, R0, R3, LSL #23
8000094c:   E1800006 ORR            R0, R0, R6
80000950:   E8BD807C LDMFD          R13!, {R2, R3, R4, R5, R6, PC}
```

```
          ovfl:
80000954:   E3A0E0FF MOV             R14, #255
80000958:   E1A0EB8E MOV             R14, R14, LSL #23
8000095c:   E186000E ORR             R0, R6, R14
80000960:   E8BD807C LDMFD           R13!, {R2, R3, R4, R5, R6, PC}

 4        int main(void) {
          main:
80000a88:   E92D400E STMFD           R13!, {R1, R2, R3, R14}
 6           float x = 1.4;
80000a8c:   E59FC020 LDR             R12, $C$FL1
80000a90:   E58DC000 STR             R12, [R13]
 7           float y = .042;
80000a94:   E59FC01C LDR             R12, $C$FL2
80000a98:   E58DC004 STR             R12, [R13, #4]
10           z = x * y;
80000a9c:   E59D0000 LDR             R0, [R13]
80000aa0:   E59D1004 LDR             R1, [R13, #4]
80000aa4:   EBFFFF84 BL              __aeabi_fmul
80000aa8:   E58D0008 STR             R0, [R13, #8]
12           return 0;
80000aac:   E3A00000 MOV             R0, #0
13       }
80000ab0:   E8BD800E LDMFD           R13!, {R1, R2, R3, PC}
```